Everything old is new again: Quoted Domain Specific Languages

Philip Wadler
University of Edinburgh

BuzzConf, Thursday 13 June 2019


WHEN SHEEP
THERE'S

How does one integrate a Domain-Specific Language and a host language?

Quotation (McCarthy, 1960)

Normalisation (Gentzen, 1935)

A functional language is a
domain-specific language
for creating
domain-specific languages

Part I

Getting started: Join queries

A query: Who is younger than Alex?

people

name	age
“Alex”	40
“Bert”	30
“Cora”	35
“Drew”	60
“Edna”	25
“Fred”	70

```
select v.name as name,
        v.age as age
from people as u,
        people as v
where u.name = “Alex” and
        v.age < u.age
```

answer

name	age
“Bert”	30
“Cora”	35
“Edna”	25

A database as data

people

name	age
“Alex”	40
“Bert”	30
“Cora”	35
“Drew”	60
“Edna”	25
“Fred”	70

$$
\begin{aligned}
&\{\text{people} = \\
&\quad [\{\text{name} = \text{“Alex”}\ ;\ \text{age} = 40\}; \\
&\quad\ \{\text{name} = \text{“Bert”}\ ;\ \text{age} = 30\}; \\
&\quad\ \{\text{name} = \text{“Cora”};\ \text{age} = 35\}; \\
&\quad\ \{\text{name} = \text{“Drew”};\ \text{age} = 60\}; \\
&\quad\ \{\text{name} = \text{“Edna”};\ \text{age} = 25\}; \\
&\quad\ \{\text{name} = \text{“Fred”}\ ;\ \text{age} = 70\}]\}
\end{aligned}
$$

A query as F# code (naive)

$$
\begin{array}{l}
\textbf{type}\ \mathsf{DB} = \{\mathsf{people} : \{\mathsf{name} : \textbf{string};\ \mathsf{age} : \textbf{int}\}\ \textbf{list}\} \\
\textbf{let}\ \mathsf{db}' : \mathsf{DB} = \textbf{database}(\text{“People”}) \\
\textbf{let}\ \mathsf{youths}' : \{\mathsf{name} : \textbf{string};\ \mathsf{age} : \textbf{int}\}\ \textbf{list} = \\
\quad \textbf{for}\ \mathsf{u}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do} \\
\quad \textbf{for}\ \mathsf{v}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do} \\
\quad \textbf{if}\ \mathsf{u.name} = \text{“Alex”}\ \ \&\&\ \ \mathsf{v.age} < \mathsf{u.age}\ \textbf{then} \\
\quad \textbf{yield}\ \{\mathsf{name} : \mathsf{v.name}; \mathsf{age} : \mathsf{v.age}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{youths}' \rightsquigarrow \\
\quad [\{\mathsf{name} = \text{“Bert”}\ ;\ \mathsf{age} = 30\} \\
\quad\ \{\mathsf{name} = \text{“Cora”};\ \mathsf{age} = 35\} \\
\quad\ \{\mathsf{name} = \text{“Edna”};\ \mathsf{age} = 25\}]
\end{array}
$$

A query as F# code (quoted)

```
type DB = {people : {name : string; age : int} list}
let db : Expr< DB > = <@ database("People") @>
let youths : Expr< {name : string; age : int} list > =
  <@ for u in (%db).people do
     for v in (%db).people do
     if u.name = "Alex" && v.age < u.age then
     yield {name : v.name; age : v.age} @>

run(youths) ⇝
  [{name = "Bert" ; age = 30}
   {name = "Cora"; age = 35}
   {name = "Edna"; age = 25}]
```

What does **run** do?

1. Normalise quoted expression
2. Translate query to SQL
3. Execute SQL
4. Translate answer to host language

Theorem

Each **run** generates one query if

A. answer type is flat (list of record of scalars)

B. only permitted operations (e.g., no recursion)

C. only refers to one database

Scala (naive)

```
val youths′ : List[{ val name : String; val age : Int}] =
  for {u ← db′.people
       v ← db′.people
       if u.name == "Alex" && v.age < u.age}
  yield new Record { val name = v.name; val age = v.age }
```

Scala (quoted)

```
val youths : Rep[ List[{ val name : String; val age : Int}] ] =
  for {u ← db.people
       v ← db.people
       if u.name == "Alex" && v.age < u.age}
  yield new Record { val name = v.name; val age = v.age }
```

Part II

Abstraction, composition, dynamic generation

Abstracting over values

```
let range : Expr< (int, int) → Names > =
  <@ fun(a, b) → for w in (%db).people do
                 if a ≤ w.age && w.age < b then
                 yield {name : w.name} @>
```

```
run(<@ (%range)(30, 40) @>)
```

```
select w.name as name
from people as w
where 30 ≤ w.age and w.age < 40
```

Abstracting over a predicate

```
let satisfies : Expr< (int → bool) → Names > =
  <@ fun(p) → for w in (%db).people do
                if p(w.age) then
                yield {name : w.name} @>

run(<@ (%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)

        select w.name as name
        from people as w
        where 30 ≤ w.age and w.age < 40
```

Dynamically generated queries

```
type Predicate =
  | Above of int
  | Below of int
  | And of Predicate × Predicate
```

```
let rec P(t : Predicate) : Expr< int → bool > =
  match t with
  | Above(a)→ <@ fun(x) → (%lift(a)) ≤ x @>
  | Below(a)→ <@ fun(x) → x < (%lift(a)) @>
  | And(t, u) → <@ fun(x) → (%P(t))(x) && (%P(u))(x) @>
```

Dynamically generated queries

$$\mathsf{P}(\mathsf{And}(\mathsf{Above}(30), \mathsf{Below}(40)))$$

$$\rightsquigarrow \texttt{<@}\ \ \textbf{fun}(x) \rightarrow (\textbf{fun}(x_1) \rightarrow 30 \leq x_1)(x)\ \ \&\&\ \ (\textbf{fun}(x_2) \rightarrow x_2 < 40)(x)\ \ \texttt{@>}$$

$$\rightsquigarrow \texttt{<@}\ \ \textbf{fun}(x) \rightarrow 30 \leq x\ \ \&\&\ \ x < 40\ \ \texttt{@>}$$

$$\textbf{run}(\texttt{<@}\ (\%\mathsf{satisfies})(\%\mathsf{P}(\mathsf{And}(\mathsf{Above}(30), \mathsf{Below}(40)))))\ \texttt{@>})$$

select w.name **as** name

from people **as** w

where $30 \leq$ w.age **and** w.age < 40

Part III

Closed quotation vs. open quotation

Dynamically generated queries, revisited

```
let rec P(t : Predicate) : Expr< int → bool > =
  match t with
  | Above(a)→ <@ fun(x) → (%lift(a)) ≤ x @>
  | Below(a)→ <@ fun(x) → x < (%lift(a)) @>
  | And(t, u) → <@ fun(x) → (%P(t))(x) && (%P(u))(x) @>
```

vs.

```
let rec P'(t : Predicate)(x : Expr< int >) : Expr< bool > =
  match t with
  | Above(a)→ <@ (%lift(a)) ≤ (%x) @>
  | Below(a)→ <@ (%x) < (%lift(a)) @>
  | And(t, u) → <@ (%P'(t)(x)) && (%P'(u)(x)) @>
```

Abstracting over a predicate, revisited

```
let satisfies : Expr< (int → bool) → Names > =
  <@ fun(p) → for w in (%db).people do
                if p(w.age) then
                yield {name : w.name} @>
```

vs.

```
let satisfies'(p : Expr< int > → Expr< bool >) : Expr< Names > =
  <@ for w in (%db).people do
       if (%p(<@ w.age @>)) then
       yield {name : w.name} @>
```

QDSL		EDSL	
$\mathsf{Expr}\langle A \to B \rangle$	✔	$\mathsf{Expr}\langle A \rangle \to \mathsf{Expr}\langle B \rangle$	✔
$\mathsf{Expr}\langle A \times B \rangle$	✔	$\mathsf{Expr}\langle A \rangle \times \mathsf{Expr}\langle B \rangle$	✔
$\mathsf{Expr}\langle A + B \rangle$	✔	$\mathsf{Expr}\langle A \rangle + \mathsf{Expr}\langle B \rangle$	✗

closed quotations

vs.

open quotations

quotations of functions

$(\mathsf{Expr}\langle\, \mathsf{A} \to \mathsf{B}\,\rangle)$

vs.

functions of quotations

$(\mathsf{Expr}\langle\, \mathsf{A}\,\rangle \to \mathsf{Expr}\langle\, \mathsf{B}\,\rangle)$

Part IV

The Subformula Principle

Gerhard Gentzen (1909–1945)

Natural Deduction — Gentzen (1935)

$\&$–I

$$\frac{\mathfrak{A} \quad \mathfrak{B}}{\mathfrak{A} \,\&\, \mathfrak{B}}$$

$\&$–E

$$\frac{\mathfrak{A} \,\&\, \mathfrak{B}}{\mathfrak{A}} \qquad \frac{\mathfrak{A} \,\&\, \mathfrak{B}}{\mathfrak{B}}$$

$\vee$–I

$$\frac{\mathfrak{A}}{\mathfrak{A} \vee \mathfrak{B}} \qquad \frac{\mathfrak{B}}{\mathfrak{A} \vee \mathfrak{B}}$$

$\vee$–E

$$\frac{\mathfrak{A} \vee \mathfrak{B} \quad \begin{matrix}[\mathfrak{A}]\\ \mathfrak{C}\end{matrix} \quad \begin{matrix}[\mathfrak{B}]\\ \mathfrak{C}\end{matrix}}{\mathfrak{C}}$$

$\forall$–I

$$\frac{\mathfrak{F}\mathfrak{a}}{\forall \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}$$

$\forall$–E

$$\frac{\forall \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}{\mathfrak{F}\mathfrak{a}}$$

$\exists$–I

$$\frac{\mathfrak{F}\mathfrak{a}}{\exists \mathfrak{x}\, \mathfrak{F}\mathfrak{x}}$$

$\exists$–E

$$\frac{\exists \mathfrak{x}\, \mathfrak{F}\mathfrak{x} \quad \begin{matrix}[\mathfrak{F}\mathfrak{a}]\\ \mathfrak{C}\end{matrix}}{\mathfrak{C}}$$

$\supset$–I

$$\frac{\begin{matrix}[\mathfrak{A}]\\ \mathfrak{B}\end{matrix}}{\mathfrak{A} \supset \mathfrak{B}}$$

$\supset$–E

$$\frac{\mathfrak{A} \quad \mathfrak{A} \supset \mathfrak{B}}{\mathfrak{B}}$$

$\neg$–I

$$\frac{\begin{matrix}[\mathfrak{A}]\\ \curlywedge\end{matrix}}{\neg \mathfrak{A}}$$

$\neg$–E

$$\frac{\mathfrak{A} \quad \neg \mathfrak{A}}{\curlywedge} \qquad \frac{\curlywedge}{\mathfrak{D}}\,.$$

Natural Deduction

$$\dfrac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \supset B}\ \supset\text{-I}^x \qquad \dfrac{A \supset B \qquad A}{B}\ \supset\text{-E}$$

$$\dfrac{A \qquad B}{A \,\&\, B}\ \&\text{-I} \qquad \dfrac{A \,\&\, B}{A}\ \&\text{-E}_0 \qquad \dfrac{A \,\&\, B}{B}\ \&\text{-E}_1$$

Proof Normalisation

$$\dfrac{\dfrac{\begin{array}{c}[A]^x\\ \vdots\\ B\end{array}}{A \supset B}\,\supset\text{-I}^x \qquad \begin{array}{c}\vdots\\ A\end{array}}{B}\,\supset\text{-E} \quad \Longrightarrow \quad \begin{array}{c}\vdots\\ A\\ \vdots\\ B\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array} \qquad \begin{array}{c}\vdots\\ B\end{array}}{A \,\&\, B}\,\&\text{-I}}{A}\,\&\text{-E}_0 \quad \Longrightarrow \quad \begin{array}{c}\vdots\\ A\end{array}$$

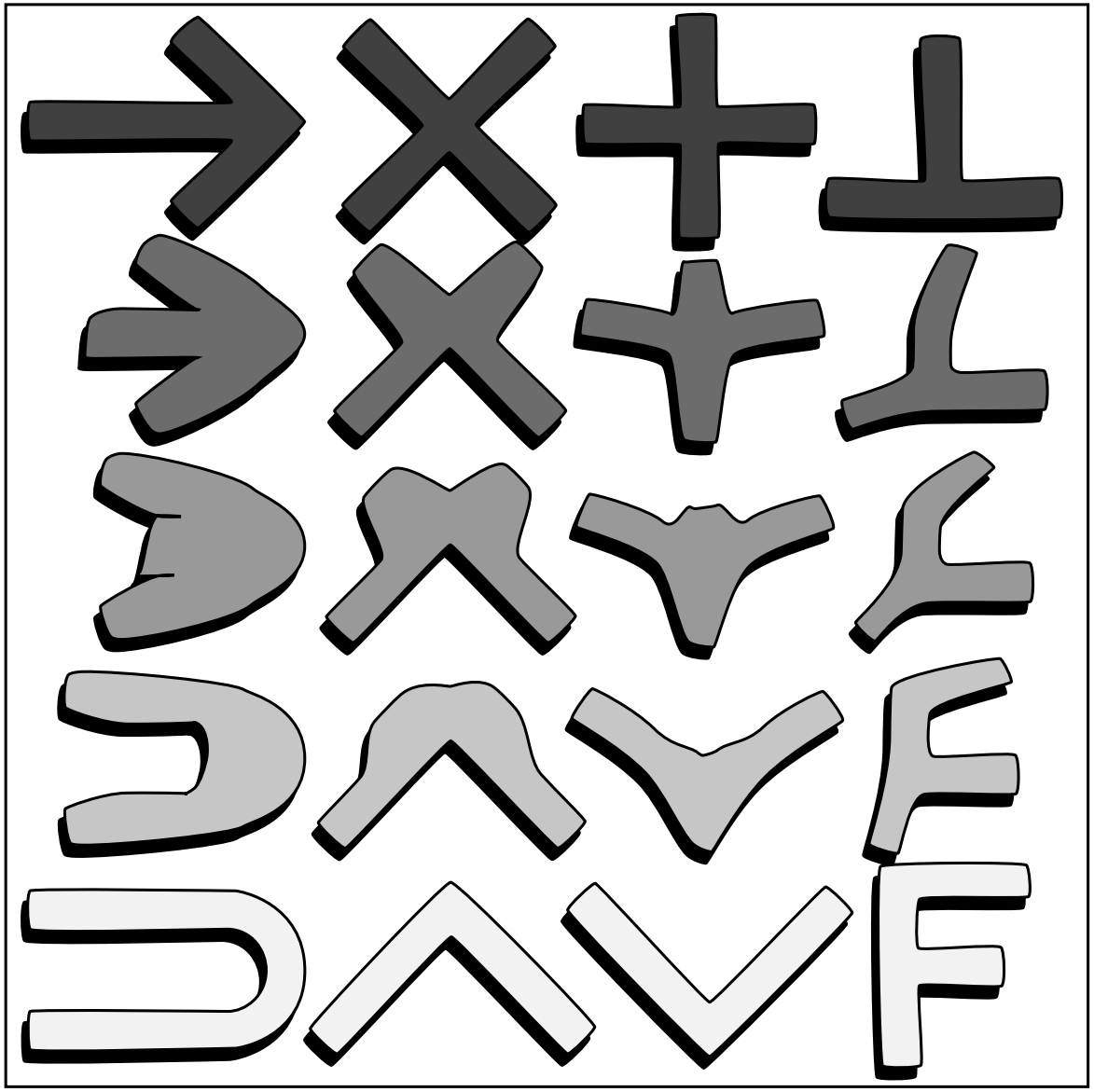
Subformula principle

Perhaps we may express the essential properties of such a normal proof by saying: it is not roundabout. No concepts enter into the proof than those contained in its final result, and their use was therefore essential to the achievement of that result.

— Gerhard Gentzen, 1935

(Subformula principle) Every formula occuring in a normal deduction in [Gentzen's system of natural deduction] of A from Γ is a subformula of A or of some formula of Γ.

— Dag Prawitz, 1965



LC'90

The Curry-Howard homeomorphism

Alonzo Church (1903–1995)

Typed λ-calculus

$$\frac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x.\, N : A \to B} \to\text{-I}^x \qquad \frac{L : A \to B \qquad M : A}{L\, M : B} \to\text{-E}$$

$$\frac{M : A \qquad N : B}{(M, N) : A \times B} \times\text{-I} \qquad \frac{L : A \times B}{\textbf{fst}\, L : A} \times\text{-E}_0 \qquad \frac{L : A \times B}{\textbf{snd}\, L : B} \times\text{-E}_1$$

Normalising terms

$$\dfrac{\dfrac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x.\, N : A \to B}\ \to\text{-I}^x \qquad \begin{array}{c}\vdots \\ M : A\end{array}}{(\lambda x.\, N)\, M : B}\ \to\text{-E} \quad \Longrightarrow \quad \begin{array}{c}\vdots \\ M : A \\ \vdots \\ N\{M/x\} : B\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ M : A\end{array} \qquad \begin{array}{c}\vdots \\ N : B\end{array}}{(M, N) : A \times B}\ \times\text{-I}}{\textbf{fst}\,(M, N) : A}\ \times\text{-E}_0 \quad \Longrightarrow \quad \begin{array}{c}\vdots \\ M : A\end{array}$$

Normalisation

$$
\begin{aligned}
(\textbf{fun}(x) \to N)\ M &\rightsquigarrow N[x := M] \\
\{\overline{\ell = M}\}.\ell_i &\rightsquigarrow M_i \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{yield}\ M)\ \textbf{do}\ N &\rightsquigarrow N[x := M] \\
\textbf{for}\ y\ \textbf{in}\ (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M)\ \textbf{do}\ N &\rightsquigarrow \textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (\textbf{for}\ y\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{if}\ L\ \textbf{then}\ M)\ \textbf{do}\ N &\rightsquigarrow \textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ [\ ]\ \textbf{do}\ N &\rightsquigarrow [\ ] \\
\textbf{for}\ x\ \textbf{in}\ (L \mathbin{@} M)\ \textbf{do}\ N &\rightsquigarrow (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N) \mathbin{@} (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{if true then}\ M &\rightsquigarrow M \\
\textbf{if false then}\ M &\rightsquigarrow [\ ]
\end{aligned}
$$

Applications of the Subformula Principle

- Normalisation eliminates higher-order functions (SQL, Feldspar)
- Normalisation eliminates nested intermediate data (SQL)
- Normalisation fuses intermediate arrays (Feldspar)

Part V

Nested intermediate data

Flat data

departments

dpt
“Product”
“Quality”
“Research”
“Sales”

employees

dpt	emp
“Product”	“Alex”
“Product”	“Bert”
“Research”	“Cora”
“Research”	“Drew”
“Research”	“Edna”
“Sales”	“Fred”

tasks

emp	tsk
“Alex”	“build”
“Bert”	“build”
“Cora”	“abstract”
“Cora”	“build”
“Cora”	“design”
“Drew”	“abstract”
“Drew”	“design”
“Edna”	“abstract”
“Edna”	“call”
“Edna”	“design”
“Fred”	“call”

Departments where every employee can abstract

```
select d.dpt as dpt
from departments as d
where not(exists(
  select *
  from employees as e
  where d.dpt = e.dpt and not(exists(
    select *
    from tasks as t
    where e.emp = t.emp and t.tsk = "abstract"
```

dpt
"Quality"
"Research"

Importing the database

$$
\begin{aligned}
\textbf{type}\ \mathsf{Org} = \{\mathsf{departments} : & \{\mathsf{dpt} : \textbf{string}\}\ \textbf{list};\\
\mathsf{employees} : & \{\mathsf{dpt} : \textbf{string}; \mathsf{emp} : \textbf{string}\}\ \textbf{list};\\
\mathsf{tasks} : & \{\mathsf{emp} : \textbf{string}; \mathsf{tsk} : \textbf{string}\}\ \textbf{list}\ \}
\end{aligned}
$$

$$\textbf{let}\ \mathsf{org} : \mathsf{Expr}\langle \mathsf{Org} \rangle = \texttt{<@}\ \textbf{database}(\text{“Org”})\ \texttt{@>}$$

Departments where every employee can do a given task

```
let expertise′ : Expr< string → {dpt : string} list > =
  <@ fun(u) → for d in (%org).departments do
                if not(exists(
                  for e in (%org).employees do
                  if d.dpt = e.dpt && not(exists(
                    for t in (%org).tasks do
                    if e.emp = t.emp && t.tsk = u then yield { })
                  )) then yield { })
                )) then yield {dpt = d.dpt} @>

run(<@ (%expertise′)("abstract") @>)
[{dpt = "Quality"}; {dpt = "Research"}]
```

Nested data

$$
\begin{array}{l}
[\{\text{dpt} = \text{“Product”};\ \text{employees} = \\
\quad [\{\text{emp} = \text{“Alex”};\ \text{tasks} = [\text{“build”}]\} \\
\quad\ \{\text{emp} = \text{“Bert”};\ \text{tasks} = [\text{“build”}]\}]\}; \\
\ \{\text{dpt} = \text{“Quality”}; \text{employees} = [\,]\}; \\
\ \{\text{dpt} = \text{“Research”};\ \text{employees} = \\
\quad [\{\text{emp} = \text{“Cora”};\ \text{tasks} = [\text{“abstract”}; \text{“build”}; \text{“design”}]\}; \\
\quad\ \{\text{emp} = \text{“Drew”};\ \text{tasks} = [\text{“abstract”}; \text{“design”}]\}; \\
\quad\ \{\text{emp} = \text{“Edna”};\ \text{tasks} = [\text{“abstract”}; \text{“call”}; \text{“design”}]\}]\}; \\
\ \{\text{dpt} = \text{“Sales”};\ \text{employees} = \\
\quad [\{\text{emp} = \text{“Fred”};\ \text{tasks} = [\text{“call”}]\}]\}]
\end{array}
$$

Nested data from flat data

```
type NestedOrg = [{dpt : string; employees :
                        [{emp : string; tasks : [string]}]}]
let nestedOrg : Expr< NestedOrg > =
  <@ for d in (%org).departments do
      yield {dpt = d.dpt; employees =
              for e in (%org).employees do
              if d.dpt = e.dpt then
              yield {emp = e.emp; tasks =
                      for t in (%org).tasks do
                      if e.emp = t.emp then
                      yield t.tsk}}} @>
```

Higher-order queries

```
let any : Expr< (A list, A → bool) → bool > =
  <@ fun(xs, p) →
        exists(for x in xs do
                if p(x) then
                yield { }) @>
let all : Expr< (A list, A → bool) → bool > =
  <@ fun(xs, p) →
        not((%any)(xs, fun(x) → not(p(x)))) @>
let contains : Expr< (A list, A) → bool > =
  <@ fun(xs, u) →
        (%any)(xs, fun(x) → x = u) @>
```

Departments where every employee can do a given task

```
let expertise : Expr< string → {dpt : string} list > =
  <@ fun(u) → for d in (%nestedOrg)
                if (%all)(d.employees,
                     fun(e) → (%contains)(e.tasks, u) then
                yield {dpt = d.dpt} @>

run(<@ (%expertise)("abstract") @>)

[{dpt = "Quality"}; {dpt = "Research"}]
```

Part VI

Compiling XPath to SQL

Part VII

Results

SQL LINQ results (F#)

Example	F# 2.0	F# 3.0	us	(norm)
differences	17.6	20.6	18.1	0.5
range	×	5.6	2.9	0.3
satisfies	2.6	×	2.9	0.3
$P(t_0)$	2.8	×	3.3	0.3
$P(t_1)$	2.7	×	3.0	0.3
expertise′	7.2	9.2	8.0	0.6
expertise	×	66.7^{av}	8.3	0.9
xp_0	×	8.3	7.9	1.9
xp_1	×	14.7	13.4	1.1
xp_2	×	17.9	20.7	2.2
xp_3	×	3744.9	3768.6	4.4

Times in milliseconds; ^{av} marks query avalanche.

Feldspar results (Haskell)

	QDSL Feldspar		EDSL Feldspar		Generated Code	
	Compile	Run	Compile	Run	Compile	Run
IPGray	16.96	0.01	15.06	0.01	0.06	0.39
IPBW	17.08	0.01	14.86	0.01	0.06	0.19
FFT	17.87	0.39	15.79	0.09	0.07	3.02
CRC	17.14	0.01	15.33	0.01	0.05	0.12
Window	17.85	0.02	15.77	0.01	0.06	0.27

Times in seconds; minimum time of ten runs.

Part VIII

Conclusion

‘Good artists copy, great artists steal’

– Pablo Picasso

“ ‘Good artists copy, great artists steal’

– Pablo Picasso ”

– Steve Jobs

“ ‘Good artists copy, great artists steal’

– Pablo Picasso ”

– Steve Jobs

EDSL	QDSL
types	types
syntax (some)	syntax (all)
	normalisation

How does one integrate a Domain-Specific Language and a host language?

Quotation (McCarthy, 1960)

Normalisation (Gentzen, 1935)

The script-writers dream, Cooper, DBPL, 2009.

A practical theory of language integrated query,
Cheney, Lindley, Wadler, ICFP, 2013.

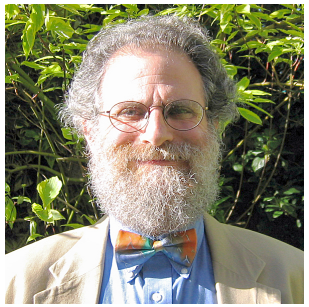
Everything old is new again: Quoted Domain Specific Languages,
Najd, Lindley, Svenningsson, Wadler, PEPM, 2016.

Propositions as types, Wadler, CACM, Dec 2015.

http://fsprojects.github.io/FSharp.Linq.Experimental.ComposableQuery/





Ezra Cooper*†, James Cheney*, Sam Lindley*,
Shayan Najd*‡, Josef Svenningsson§, Philip Wadler*
*University of Edinburgh, †Qumulo, ‡Google, §Chalmers & HiQ